

Efficient Spatial Keyword Search in Trajectory Databases

Gao Cong [†]Hua Lu [§]Beng Chin Ooi [‡]Dongxiang Zhang [‡]Meihui Zhang [‡][†]School of Computer Engineering, Nanyang Technological University[§]Department of Computer Science, Aalborg University[‡]School of Computing, National University of Singapore

gaocong@ntu.edu.sg

luhua@cs.aau.dk

{ ooibc | mhzhang | dxzhang }@comp.nus.edu.sg

ABSTRACT

An increasing amount of trajectory data is being annotated with text descriptions to better capture the semantics associated with locations. The fusion of spatial locations and text descriptions in trajectories engenders a new type of top- k queries that take into account both aspects. Each trajectory in consideration consists of a sequence of geo-spatial locations associated with text descriptions. Given a user location λ and a keyword set ψ , a top- k query returns k trajectories whose text descriptions cover the keywords ψ and that have the shortest match distance. To the best of our knowledge, previous research on querying trajectory databases has focused on trajectory data without any text description, and no existing work has studied such kind of top- k queries on trajectories. This paper proposes one novel method for efficiently computing top- k trajectories. The method is developed based on a new hybrid index, cell-keyword conscious B⁺-tree, denoted by B^{ck}-tree, which enables us to exploit both text relevance and location proximity to facilitate efficient and effective query processing. The results of our extensive empirical studies with an implementation of the proposed algorithms on BerkeleyDB demonstrate that our proposed methods are capable of achieving excellent performance and good scalability.

1. INTRODUCTION

With the increasing popularity of crowdsourcing, as well as the advancements and miniaturization of handheld devices with GPS receivers, massive amount of data that are geo-tagged or associated with text information are being generated at an unprecedented scale. For example, crowdsourcing of motion trajectories is applied to generate the Open map systems (e.g., openstreetmap.org and waze.com).

Users have crowdsourced huge volumes of trajectory data that are annotated with keywords or text descriptions. In such datasets, a trajectory is composed of a sequence of places and line segments connecting these places. The places in a trajectory, captured as spatial locations, are often associated with text descriptions. Figure 1 shows an example of a trajectory. Such trajectories come from various sources, and we name just a few in the following: 1) In many GPS-trajectory-sharing websites (e.g., Moun-

tain Bike: www.bikely.com, GPS sharing: www.gpssharing.com, GPSies: www.gpsies.com, and Geolife [31]), people upload their travel routes. To record their journeys or share life experiences with others, they often attach texts and multimedia content (e.g., photos) as annotations to the places in their trajectories. 2) In location-based social network services (e.g., FourSquare), each place is associated with tags and users can check in such places. The check-in sequence of a user in a period forms a trajectory. The places can points of interests of any kind, e.g., restaurants, shops, and thus, the trajectories can be of various types, such as travel trajectories and daily life trajectories. 3) Trajectories with text descriptions can be extracted from travel itineraries [16], as well as Flickr photos [19].

Such publicly accessible datasets serve as an informative repository to users. A user may want to find others' travel routes that are relevant to his/her interests and that have a short travel distance. Motivated as such, we consider queries that search previously explored routes of places that satisfy a user's interests or needs, expressed as a set of keywords, and that may also lead to the shortest total traveling distance. The results of such a query exploit the collective intelligence of crowdsourcing.

In addition, users may be interested in learning the daily life experience of others. For example, from relevant social network applications, it is easy to derive a shopping trajectory database, where each place (corresponding to a shop) in a user-generated trajectory is associated with the items bought by the user at that place. Such a user-generated trajectory indicates the user's preferences. Suppose that a user has a shopping list of product names. She would like to see the routes of other users who buy all the items on the list, and the traveling distance from her starting location along this route that is the minimum.

The fusion of spatial locations and text descriptions in trajectories demands efficient processing of queries that involve both attributes. Indeed, the aforementioned GPS sharing websites already support a type of queries related to both text and locations, namely the keyword range queries, to help users share, browse and search GPS trajectories. They allow users to specify a region and a set of keywords, and return the trajectories that are inside the query region and contain the set of query keywords. However, the algorithms used are not publicized, and the response for answering such queries in these websites is very slow.

Existing research on querying trajectory database has focused on trajectory data without any text description. For example, a k -Nearest Neighbor query [13] returns the k nearest moving object trajectories to a given query point based on the minimum distance from the query point to a trajectory. Querying trajectory data is time consuming and therefore, indexes such as the R-tree and its optimized versions for trajectories have been used.

To the best of our knowledge, no publication considers query-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...\$10.00.

ing trajectories that are composed of a sequence of geo-locations associated with text descriptions.

In this paper, we introduce a new problem: the top- k spatial keyword query (TkSK) on trajectories. Given a large database of trajectories, a TkSK query consists of a spatial element (query location) and a set of keywords, and it returns the top- k trajectories with the shortest match distance. The match distance is measured by the sum of two distances: the length of a sub-trajectory covering all query keywords, and the distance from the query location to the start location of the sub-trajectory. It is a challenge to efficiently answer the TkSK query on trajectories associated with text.

To this end, we propose a novel solution with the following features. First, we develop a new index for trajectories, called cell-keyword conscious B^+ -tree, denoted by B^{ck} -tree. B^{ck} -tree integrates spatial information captured by location keys generated by adaptive cells and text information such that it enables simultaneous application of both spatial proximity and keyword matching in query processing. The B^{ck} -tree is efficient for queries as well as updates, and it is adaptive to varying workloads. Further, with the use of the B^+ -tree that is available in all mainstream DBMSs, our proposed solution can be easily grafted onto existing database systems. Second, based on the B^{ck} -tree, we develop an algorithm for choosing candidate trajectories that are close to the query location and contain the query keywords, and thus are more likely to be the results of a TkSK query. Third, we propose a linear time algorithm, called Match, for efficiently computing the match distance between a query and a candidate trajectory, which contrasts with a straightforward method that takes quadratic time.

Since no baseline algorithms exist for processing TkSK queries, we also develop four baseline algorithms. They all use the proposed algorithm Match for computing the matching distance. They differ in their ways of finding candidate trajectories: 1) The first one uses the inverted list index to choose the trajectories containing all query words. 2) The second uses the R-tree to retrieve nearby trajectories. 3) The third is based on the IR-tree [10], treating each trajectory as a whole to retrieve nearby trajectories containing query keywords. 4) The fourth extends the TB-tree [22], an existing index for trajectories, to incorporate the text information organized in an inverted index, and uses the extended TB-tree to retrieve candidate trajectories.

In summary, the paper's contributions are threefold. First, we introduce and formalize a new type of queries on trajectory data that are associated with words. Second, we propose a novel solution for efficiently processing TkSK queries. The proposed solution consists of a new index structure B^{ck} -tree for trajectories associated with words, an approach to computing the minimum match distance between a trajectory and a query, and a top- k query processing algorithm. The proposed solution can be implemented on top of existing DBMSs cost-effectively. We also explore other ways of answering TkSK queries as baseline methods. Third, with an implementation of the B^{ck} -tree based algorithm on BerkeleyDB, we conduct an extensive experimental study, which includes a comparison with the four baselines. The experimental results demonstrate the efficiency and scalability of our proposed solution.

The rest of the paper is organized as follows. Section 2 defines the TkSK query. Section 3 presents the baseline algorithms. Section 4 details our solution for processing the TkSK query. Section 5 reports the experimental study. Section 6 reviews related work. Section 7 concludes this paper.

2. PROBLEM STATEMENT

In this section, we give the problem statement and provide necessary definitions and background.

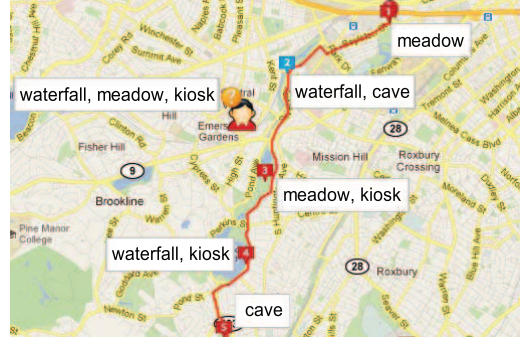


Figure 1: Example

Data Let \mathcal{D} be a dataset in which each object is a trajectory.

Definition 1: Trajectory

Each trajectory $\mathcal{T} \in \mathcal{D}$ is defined as a sequence of places (points of interests) $\mathcal{T} = L_1, \dots, L_i, \dots, L_n$. \square

Each place L is represented by a pair $(L.\lambda, L.\psi)$, where $L.\lambda$ represents a geo-spatial point location and $L.\psi$ denotes a set of keywords (e.g., the description about the place).

We denote the union of the text description of each place in trajectory \mathcal{T} by $\mathcal{T}.\psi = \bigcup_{i=1}^n \mathcal{T}.L_i.\psi$.

Definition 2: Sub-Trajectory and Contain

We define a sub-trajectory as a subsequence from place s to place e of trajectory \mathcal{T} as $\mathcal{T}.L_s^e$, $s, e \in [1, n]$, $s \leq e$. Given two sub-trajectories $\mathcal{T}.L_{s_1}^{e_1}$ and $\mathcal{T}.L_{s_2}^{e_2}$, we say that $\mathcal{T}.L_{s_1}^{e_1}$ contains $\mathcal{T}.L_{s_2}^{e_2}$ if $s_1 \leq s_2$ and $e_1 \geq e_2$. \square

We denote the union of the text description of each place in sub-trajectory $\mathcal{T}.L_s^e$ by $\mathcal{T}.L_s^e.\psi = \bigcup_{i=s}^e \mathcal{T}.L_i.\psi$.

Query A spatial keyword query $q = \langle \lambda, \psi \rangle$ has two components, where $q.\lambda$ is a spatial location and $q.\psi$ is a set of keywords. The location descriptor $q.\lambda$ specifies the location preference of a user, and $q.\psi$ indicates the preference of a user on the keywords of objects.

Definition 3: Match

We say that a trajectory \mathcal{T} matches a query q if the following condition is satisfied.

$$q.\psi \subseteq \mathcal{T}.\psi$$

Similarly, we say that a sub-trajectory $\mathcal{T}.L_s^e$ matches a query if $q.\psi \subseteq \mathcal{T}.L_s^e.\psi$. \square

Intuitively, we say that a trajectory \mathcal{T} matches a query q if all the keywords of the query are contained in the text of the trajectory.

Definition 4: Minimum Match

We say that a sub-trajectory $\mathcal{T}.L_s^e$ is a minimum match of a query q if (1) $\mathcal{T}.L_s^e$ matches q ; and (2) no sub-trajectory of $\mathcal{T}.L_s^e$ matches q . \square

Example 1: Refer to Figure 1. A traveller wants to find a route in which she can see waterfall, panda and kiosk. There are two minimum matches to the query $\{\text{waterfall, meadow, kiosk}\}$: $L_2 - > L_3$ and $L_3 - > L_4$. Note that $L_2 - > L_3 - > L_4$ is a match but it is not a minimum match. \square

Definition 5: Match Distance

If a sub-trajectory $\mathcal{T}.L_s^e$ matches a query q , the match distance $\text{matchDist}(q, \mathcal{T}.L_s^e)$ is defined as follows:

$$\text{matchDist}(q, \mathcal{TR}.L_s^e) = \min\{\text{Dist}(q, \mathcal{TR}.L_s), \text{Dist}(q, \mathcal{TR}.L_e)\} \\ + \sum_{i=s}^{e-1} \text{Dist}(\mathcal{TR}.L_i, \mathcal{TR}.L_{i+1}),$$

where $\text{Dist}(\cdot, \cdot)$ is the Euclidean distance between two locations.

If a sub-trajectory $\mathcal{TR}.L_s^e$ does not *match* a query q , the match distance is defined as ∞ . \square

Definition 6: Minimum Match Distance

If a trajectory \mathcal{TR} **matches** a query q , the minimum match distance $\text{minMatchDist}(q, \mathcal{TR})$ is defined as follows:

$$\text{minMatchDist}(q, \mathcal{TR}) = \min_{(s,e)} (\text{matchDist}(q, \mathcal{TR}.L_s^e)), \\ \text{s.t., } \mathcal{TR}.L_s^e \text{ is a minimum match of } q.$$

\square

Example 2: Consider the example in Figure 1. When we take her current location $q.\lambda$ into account, sub-trajectory $L_3 - > L_4$ is the one with the minimum Match Distance. \square

Definition 7: Top- k Spatial Keyword query (TkSK)

Given a trajectory set \mathcal{D} , a top- k spatial keyword query (TkSK) with $q = \langle \lambda, \psi \rangle$ returns from \mathcal{D} k trajectories that have the smallest minimum match distances with respect to q , each associated with the start and end place indexes that yield the minimum match distance.

Formally, a TkSK query returns a set $\text{Ans}(\mathcal{D}, q)$ of k triples (t, s, e) , where $t \in \mathcal{D}$, $1 \leq s \leq e \leq |t|$, such that

1. $|\text{Ans}(\mathcal{D}, q)| = |\pi_1(\text{Ans}(\mathcal{D}, q))| = k$, where $\pi_1(\cdot)$ denote the projection on the first attribute of a set of triples of the format (t, s, e) .
2. $\forall (t, s, e) \in \text{Ans}(\mathcal{D}, q)$, $\text{matchDist}(q, t.L_s^e) = \text{minMatchDist}(q, t)$;
3. $\forall (t, s, e) \in \text{Ans}(\mathcal{D}, q)$, $\forall t' \in \mathcal{D} \setminus \pi_1(\text{Ans}(\mathcal{D}, q))$, the following inequality holds: $\text{minMatchDist}(q, t) \leq \text{minMatchDist}(q, t')$.

Intuitively, the answer to the query consists of k sub-trajectories from k distinct trajectories whose minimum match distances to query q are the smallest. \square

3. BASELINE ALGORITHMS

No baseline algorithm exists for the top- k spatial keyword queries on trajectory data. We develop four baseline algorithms. The four baseline algorithms constitute a contribution to the problem of processing the top- k spatial keyword queries in that they explore the possibility of using existing index techniques for the new problem. The four baseline algorithms employ the algorithm (presented in Section 4.3) for computing match distance. Baseline 4 is lengthy and is described in Appendix. The baseline algorithms act as background for better understanding of the problem and its complexity.

3.1 Baseline 1: IF

The first baseline, IF, uses Inverted File as the index structure. Specifically, it aggregates the text description associated with each place in a trajectory to get a set of words of the trajectory, and then builds inverted file for all the trajectories.

The idea of the IF algorithm is to use the inverted file to filter out the trajectories that do not contain all the keywords of query q , *i.e.*, finding the set of trajectories T_m that match the query. Then for each trajectory in T_m , we compute its matchDistance to the query using the algorithm presented in Section 4.3 and find the top- k trajectories.

3.2 Baseline 2: RT

The second baseline, RT, uses an R-tree [14] as the index structure. Specifically, it aggregates the MBR associated with each place in a trajectory to get the MBR of the trajectory, and then uses an R-tree to index all the trajectories. For each trajectory, this baseline uses a separate index structure to organize the text description associated with places of the trajectory as the component 2 in Section 4.1.

Given a query q , the baseline uses the R-tree to find the nearest trajectory incrementally. For each nearest trajectory, we check if it matches the query keywords. If yes, we compute its matchDistance to the query using the algorithm in Section 4.3. In the process, the algorithm keeps track of the minimum match distance of the current k th trajectory, denoted by *threshold*. For a newly “seen” trajectory with spatial distance *dist* to query q , if the score *dist* exceeds *threshold*, the algorithm stops since it is guaranteed that all “unseen” trajectories will not have smaller match distance than the current k ’th trajectory (and thus cannot be in the result). Note that *dist* is a lower bound of the minimum match distance.

3.3 Baseline 3: IRT

The third baseline, IRT, uses the IR-tree [10] as the index structure, which is used to index spatial Web objects. The IR-tree is essentially an R-tree [14] extended with inverted files [33]. Each leaf node in the IR-tree contains a number of entries of the form $(p, p.\lambda)$, where p refers to the identifier of a spatial object, and $p.\lambda$ is the bounding rectangle of p . Each leaf node also contains a pointer to an inverted file for the text descriptions of the objects stored in the node. Each non-leaf node R in the IR-tree contains a number of entries of the form $(cp, rect, cp.di)$ where cp is the address of a child node of R , $rect$ is the MBR of all rectangles in entries of the child node, and $cp.di$ is the identifier of a pseudo text description of the child node. The pseudo text description is a union of all text descriptions in the entries of the child node. Each non-leaf node also contains a pointer to an inverted file for the pseudo text descriptions of its child nodes. The pseudo text description enables us to prune a node (and the subtree under the node) if it does not cover all the query keywords.

To use the IR-tree [10] to organize the trajectories, we aggregate the MBR associated with each place in a trajectory to get the MBR of the trajectory; similarly we get the set of words of the trajectory by aggregating the text description of each place.

We adapt top- k algorithm presented in [10] that is based on the best-first search to find the top- k trajectories. A priority queue U is used to keep track of the nodes and trajectories that have yet to be visited. The values of $\text{minDist}(q, \cdot)$, which is the minimum Euclidean distance between q and a trajectory (or a node), are used as the keys. Note that the key used for a trajectory in U is not the match distance, but a loose lower bound of the match distance between query and trajectories in a node. It is used to choose which node to visit next and when to terminate the algorithm. When deciding which node to visit next, the algorithm picks the node CN with the smallest $\text{minDist}(q, CN)$ value in the set of all nodes that have yet to be visited. The algorithm terminates when the match distance of k th trajectory is smaller than the key of first element in U . Algorithm 1 shows the pseudo-code.

3.4 Discussion

The first baseline IF uses the text information to prune the search space without utilizing the spatial information to speed up. The second baseline RT uses the spatial information to guide the search for results without utilizing the text information.

Different from the first two baselines, the baseline IRT (and the

Algorithm 1: IRT (query q , Tree root $root$, Integer k)

```

1  $V \leftarrow$  new max-priority queue of  $k$  elements of  $\infty$ ;
2  $U \leftarrow$  new min-priority queue;
3  $U.Enqueue(root, 0)$ ;
4 while  $U$  is not empty do
5    $e \leftarrow U.Dequeue()$ ;
6   if  $(\minDist(q.\lambda, e.\lambda) \geq V[k])$  then
7     break while-loop;
8   if  $e$  is a trajectory then
9     update  $V$  by  $(e, Match(q, e, V[k]))$ ;
10  else //  $e$  points to a child node
11    read the node  $CN$  of  $e$ ;
12    read the posting lists of  $CN$  for keywords in  $q.\psi$ ;
13    for each entry  $e'$  in the node  $CN$  do
14      if  $q.\psi \subseteq e'. and  $\minDist(q.\lambda, e'. then
15         $U.Enqueue(e', \minDist(q.\lambda, e'.;
16 return  $\{V\}$ ; // top- $k$  results$$$ 
```

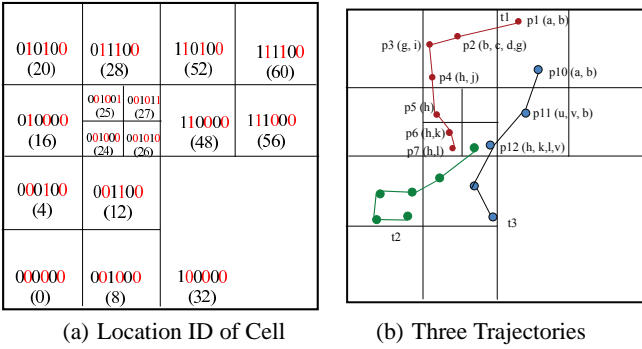


Figure 2: Example

baseline given in Appendix) is able to make use of both text information and distance information to prune the search space. However IRT faces the following challenges: The MBR of a trajectory can be much larger than the real geographical space of places in the trajectory, and thus the MBRs of nodes in the IR-tree have large overlapping. The text description of a trajectory is the aggregation of the descriptions of all places in the trajectory. Hence, the overlapping of text descriptions between nodes with large overlapping MBRs is also large. Thus the pruning power of the text information associated in the IR-tree nodes might be limited.

4. PROPOSED ALGORITHMS FOR QUERY PROCESSING

Section 4.1 presents the proposed index B^{ck}-tree. Based on the index, we present the incremental expansion algorithm for finding candidate trajectories in Section 4.2. Section 4.3 presents an algorithm for matching a candidate trajectory with a query.

4.1 Proposed Index: B^{ck}-tree

Ideally, we can index trajectories associated with text information to enable pruning search space by utilizing both spatial dis-

tance and keyword information for efficient query processing. It is, however, challenging to develop indexes to meet the complexities of trajectories associated with text information. To this end, we propose an index, called cell-keyword conscious B⁺-tree, denoted by B^{ck}-tree, which comprises two components.

1) Component 1 is used to locate the IDs of trajectories that are close to the query location and contain all the keywords. It is used to organize the segment-level information of trajectories.

2) Component 2 is used to compute the minimum match distance of a selected trajectory to query q . It is used to organize the detailed information of each trajectory.

Component 1: We divide the spatial region of dataset \mathcal{D} into quad cells of various sizes to generate location codes. The ID of each cell can be generated by using the bit-interleaving method [1]. If a quad cell consists of a set of uniform cells, the minimum ID of the set of cells will be the ID of the quad cell. Figure 2(a) shows an example. Based on the cell division, we build a B⁺-tree to index trajectories together with their text descriptions. Each leaf entry contains three elements:

- wordID: it denotes the ID of a word in the trajectory database.
- cellID: it denotes the ID of a cell that contains a wordID.
- posting list: it is a sequence of trajectory identifiers for each wordID and cellID, *i.e.*, the list of trajectories in cell cellID that contain word wordID.

In the index, the entries are organized first by the word ID, and next by the cell ID. Hence, the posting lists for the same word are organized together, and posting lists of nearby cells for the same word are together. This enables visiting nearby cells for a word by following the pointers between leaf nodes of B⁺-tree.

All distinct words in the text description of the trajectory database constitute a vocabulary, and each word has a wordID. We proceed to explain the other two elements, cellID and posting list.

cellID: The cellID element aims to integrate the spatial information and text information of trajectories. We partition the index space into cells, and thus one trajectory may span multiple cells. The sizes of cells are not fixed. We set the size of a cell such that the number of trajectories in a cell is smaller than a threshold ξ . Note that the empty cells are not indexed.

posting list: Given a set of query words, we need to check if a cell contains a trajectory that covers all the query keywords. To meet the need, for each wordID w , and cellID c , a posting list is a sequence of identifiers of the trajectories such that part of the trajectory or the whole trajectory falls in cell c , and the text description associated with the trajectory segment in c contains word w . Such a design enables associating the cell ID, which represents the spatial information of a trajectory segment, with the text information of the segment.

Example 3: In Figures 2(a)-2(b), for cell 52, we generate one entry $(a, 52, \langle t1, t3 \rangle)$ since the fragments of trajectories $t1$ and $t3$ in cell 52 contain word a . Similarly, we generate another entry $(b, 52, \langle t1, t3 \rangle)$ for cell 52. As another example, for cell 28 two example entries (out of totally 7) include $(g, 28, \langle t1 \rangle)$, $(b, 28, \langle t1 \rangle)$. Here we do not include the detailed information on which places contain a specific word for a trajectory. It is also noteworthy that empty cells are not indexed. \square

However, the above design will be problematic at query time when a trajectory spans multiple cells, and individual fragment does not contain all the query words, but several fragments together match all the query words (which will become clear in the next section). A simple fix is to associate each cell with all the words of a trajectory. However, this significantly increases the space cost.

We next present a carefully designed mechanism that needs less space while returning the correct results. Suppose that a trajectory \mathcal{TR} falls in m cells. We denote a trajectory fragment as \mathcal{TR}_i , $i \in [1, m]$, where \mathcal{TR}_i is adjacent to \mathcal{TR}_{i+1} in the trajectory. We associate words for each trajectory fragment as follows.

- If i is odd, the set of words for fragment \mathcal{TR}_i is the union of words in the places in the fragment.
- If i is even, the set of words for the fragment \mathcal{TR}_i will be $\bigcup_{j=1}^{\min\{i+1, m\}} \mathcal{TR}_j \cdot \psi$, where $\mathcal{TR}_j \cdot \psi$ is the union of words in the places in fragment \mathcal{TR}_j .

For example, consider trajectory $t2$ in Figure 3 with three segments in three cells. Each of the three segments contains a term. According to the proposed mechanism, we associate segment 1 with a , segment 2 with a, b, c and segment 3 with c .

This method can guarantee the correctness of the proposed algorithms and we prove this in Lemma 2. Note that one cell can contain multiple fragments of the same trajectory, and the aforementioned method is equally applicable.

Component 2: We use a B^+ -tree to organize the place information and the associated keywords in all the trajectories. The text description for a place can be either short or long. We use inverted list for each trajectory. Each entry consists of three elements: *trajectory ID*, *word ID*, *list of place IDs in the trajectory*, where trajectory ID and word ID compose the key of the B^+ -tree. Note that the inverted file is the most efficient index for text information retrieval [33].

We discuss the updating process of the B^{ck} -tree in the Appendix.

Algorithm 2: IE(query q , result size k)

```

1  $V \leftarrow$  new min-priority queue of  $k$  element of  $\infty$ ; // maintain top-k
  trajectories
2  $i \leftarrow 0$ ;
3 while true do
4    $rq_i \leftarrow$  compute a range radius;
5    $R_i \leftarrow$  construct a range with  $q$  as the center and  $rq_i$  as extension;
6   if  $i \neq 0$  then
7      $R_i \leftarrow R_i - R_{i-1}$ ;
8    $A \leftarrow \text{CTR}(q, R_i)$ ; // See Procedure CTR
9   for each trajectory  $t$  in  $A$  do
10    read post lists of  $t$  for keywords in  $q$ ;
11     $dist \leftarrow \text{Match}(q, t, V[k])$ ; // See Section 4.3
12    if  $V[k] > dist$  then  $V.add(dist, t)$ ;
13    if  $V[k] < rq_i$  then break;
14     $i \leftarrow i + 1$ ;
15 return top- $k$  trajectories in  $V$ ; // top- $k$  results

```

The proposed index solution B^{ck} -tree can be implemented using DBMSs that support the B^+ -tree, and is update friendly. It enables designing algorithms for processing TkSK queries that are able to prune the search space using both types of information. In addition to the TkSK queries, it also support other types of queries containing a keyword component and a spatial component, e.g., finding trajectory containing a set of keywords within a region. Different from the IRT baseline that takes each trajectory as a whole and associates text information with the trajectory, in the proposed index we use cells to divide trajectories into segments and design an effective mechanism to associate keywords to segments. For example, in Figure 3, given a top-1 query q with keyword $b, t1$ is the answer. If we use the proposed word association mechanism, we can prune $t2$ since the segment in the first cell is associated with

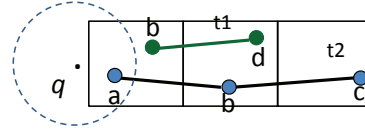


Figure 3: Associating words with trajectory segments

word a only. However, in the IRT, we cannot prune $t2$ since it takes the trajectory as a whole. As another example, if q is to find trajectories whose segment contains b and falls in the circle, we can prune $t2$ while $t2$ cannot be pruned if treated as a whole.

4.2 Incremental Expansion Algorithm (IE)

We compute top- k trajectories by iteratively performing range queries with an incrementally expanded search region on the B^{ck} -tree until the top- k matching trajectories are retrieved. The Incremental Expansion algorithm (IE) is outlined in Algorithm 2. The algorithm IE first initializes a priority queue to maintain top- k results. We construct a range query with q as the center and a query dependent rq_0 as the extension (lines 4-5).

To compute extension rq_0 , we take into account both keyword information and spatial information. Let $p(q, \psi)$ be the probability of containing q, ψ as the keyword set of a trajectory of \mathcal{D} . We estimate the probability by $p(q, \psi) = \prod_{w \in q, \psi} p(w)$, where $p(w)$ can be estimated using the maximum likelihood estimation, i.e., the probability of a trajectory in dataset \mathcal{D} that contains the word. We compute rq_0 by

$$rq_0 = \sqrt{\frac{k \times L}{\pi \times |\mathcal{D}| \times p(q, \psi)}},$$

where L is the area size of the whole region, since a region of area size $\pi \times rq_0^2$ would probabilistically contain segments of k trajectories that contain all query keywords if the trajectories are uniformly distributed in the whole region.

For a trajectory in the range, we check if it contains all the query keywords, and compute its matching distance by invoking function CTR (Candidate Trajectory Retrieval, to be presented shortly). For each returned candidate trajectory t in A , the algorithm invokes algorithm $\text{Match}(q, t, V[k])$ (See section 4.3) to compute the minimum match distance between q and t . If the match distance of the k th result is larger than rq_i , it is safe to terminate the algorithm because the algorithm has considered all the trajectories that can possibly be in the top- k results. Otherwise, we compute a new range $rq_i = rq_{i-1} + \tau$, where τ is the side length of the smallest quad cell in the index. We also tried other options, e.g., the average side length. However, the performance of such options is worse in general. We then retrieve trajectories in the region formed by radius rq_i , but not included in the region formed by rq_{i-1} .

We proceed to present procedure CTR, which checks if a trajectory in the given range R contains all the query keywords. CTR processes query keywords one by one. For the first query keyword, we find the trajectories that contain the query keyword, and intersect with the given query range R . Recall that B^{ck} -tree organizes the list of trajectories by word ID and then by cell ID. This enables us to retrieve those trajectories that contain the query keywords and fall in certain cells. For each subsequent query keyword, we filter out trajectories that do not contain the query word by scanning the corresponding cells.

The candidate trajectory retrieval (CTR) algorithm is outlined in Procedure CTR. It takes two arguments: query q and the given region R . It first computes the intervals of cell IDs that are covered by the query region R (line 2). The algorithm proceeds to process

Procedure CTR(q, R)

```

1  $A \leftarrow$  new array; // maintain the trajectories to be checked
2  $I \leftarrow$  Compute the intervals  $\{(sid_j, eid_j)\}$  of  $R$ ; // start cell id
    $sid_j$  and end cell id  $eid_j$ 
3 for each keyword  $w_i$  ( $i = 1, \dots, |q|$ ) do
4   for each interval  $I_j = (sid_j, eid_j)$  in  $I$  do
5      $(C_j, I'_j) \leftarrow$  getCellInterval( $w_i, I_j$ );
6     if  $I'_j = \emptyset$  then
7       remove  $I_j$  from  $I$ ;
8       continue;
9     else
10      update  $I$  with  $I_j$ ;
11     if  $i = 1$  then
12       for each cell  $c$  in  $C_j$  do
13         add trajectories in  $c$  to  $A$ ;
14     else
15       removes trajectories in  $A$  that are not covered by any cell
          $c$  in  $C_j$ 
16 return  $A$ ;

```

each query word w_i (lines 3–15). For each interval, it returns the trajectories that contain keyword w_i and fall in the interval. Function `getCellInterval(.)` returns in C_j those cells that contain word w_i and fall in the interval I_j . The function is implemented by following pointers between leaf nodes of B^+ -tree, and the jump technique [23] is used to optimize the implementation by jumping over pages. If interval $I_j = (sid_j, eid_j)$ does not contain any cell containing word w_i , we remove the interval from consideration (lines 6–8). Function `getCellInterval(.)` also returns a smaller interval I'_j if the interval covered by cells in C_j is smaller than I_j . We use I_j to update the interval boundary sid_j and eid_j (line 10). For the first keyword w_1 , we add the trajectories that contain word w_1 and are in the region R to the set of candidate trajectories A (lines 11–12). For each of subsequent keyword, the algorithm filters the trajectories in A that do not contain the keyword (line 15). In the implementation, we organize trajectories in A by cell ID and filter trajectories in a cell if the cell does not contain a query word.

We process query words in the ascending order of their frequencies, i.e., infrequent words are processed first. The reason is that infrequent keywords are more likely to prune trajectories.

Before we prove the correctness of the proposed algorithm, we first present a lemma.

Lemma 1: Consider query q and a trajectory \mathcal{TR} that falls in m cells. We denote a trajectory fragment as \mathcal{TR}_i , $i \in [1, m]$ and the cell containing trajectory fragment \mathcal{TR}_i as $cell(\mathcal{TR}_i)$. If the minimum match of \mathcal{TR} for q that results in the minimum match distance follows in cells $[c_1, c_2]$, $1 \leq c_1 \leq c_2 \leq m$, we have $\minMatchDist(q, \mathcal{TR}) \geq \max_{c_j \in [c_1, c_2]} (\minDist(q, c_j))$.

Proof Sketch: Based on triangular inequality, we know that $\minMatchDist(q, \mathcal{TR}) \geq Dist(q, \mathcal{TR}_L)$, where \mathcal{TR}_L is a place in the sub-trajectory of \mathcal{TR} that is a minimum match of query q . It is easy to see that $\minDist(q, c_j)$ is not larger than $Dist(q, \mathcal{TR}_L)$ where L is in cell c_j . We complete the proof. \square

According to Lemma 1, the minimum match distance of a trajectory to the query is larger than the distance from query to any cell that contain parts of the matching trajectory. We are now ready to present the correctness of the IE.

Lemma 2: The Incremental Expansion algorithm guarantees to find top- k trajectories using B^{ck} -tree that employs the method (in

Section 4.1) of associating the words of the trajectory with the different segments.

Proof Sketch: Suppose that a trajectory \mathcal{TR} falls in m cells. We denote a trajectory fragment as \mathcal{TR}_i , $i \in [1, m]$. Two cases cover all the possibilities that \mathcal{TR} is one of the top- k results.

Case 1: if \mathcal{TR}_i contains all the query words and the cell containing \mathcal{TR}_i is in the range of the match distance of k th trajectory in the current result set (i.e., $\minDist(cell(\mathcal{TR}_i)) > \minMatchDist(\mathcal{TR}, q)$) the trajectory \mathcal{TR} will be retrieved and we compute its match distance. In this case, algorithm IE will not miss the trajectory \mathcal{TR} if it is a result. Note that if the cell containing \mathcal{TR}_i is not in the range, \mathcal{TR}_i cannot be matching part of top- k trajectory.

Case 2: we next consider the case that none of the sub-trajectories contains all the query keywords, but the trajectory contains all the query keywords. We first consider that two subtrajectories in two adjacent cells cover the query keywords. The method of associating keywords with cells make sure that one of the two subtrajectories in the two cells will be associated with at least the keywords of both subtrajectories. According to Lemma 1, the distance from query to the cell containing a sub-trajectory associated with all keywords of the two adjacent cells must be smaller than the match distance between the trajectory and query. This grants that algorithm IE will not miss the trajectory \mathcal{TR} if it is a result. Similarly, when more than 2 adjacent cells together cover the query keywords, at least one of the subtrajectories of \mathcal{TR} in these cells contain all the keywords of these sub-trajectories according to the method of associating keywords to sub-trajectories. Thus, algorithm IE will not miss the trajectory \mathcal{TR} if it is a result.

The two cases cover all the possibilities that a trajectory can be a top- k result. Therefore, Algorithm IE is correct and complete. \square

4.2.1 Cost Analysis of IE Algorithm

First of all, it is noteworthy that our incremental expansion algorithm (IE in Algorithm 2) has an asymptotically equivalent effect of a window search through the specific B^+ -tree index that is also known as a linear quadtree. In particular, such an equivalent query is centered at query location $q.\lambda$ and its window size is bounded by the place L_{last} that our algorithm fetches as the last place on the trajectory that contributes to the k -th minimum match distance.

To make the analysis clear, we assume that k is 1, i.e., we only get the top-1 trajectory with minimum match distance. Let the distance from $q.\lambda$ to L_{last} through the trajectory, i.e., the corresponding minimum match distance, be $Dist(q.\lambda, L_{last})$. This matching distance can be used as the half window size in the aforementioned equivalent window query.

The matching distance $Dist(q.\lambda, L_{last})$ is not a Euclidean distance since we work with trajectories. Nevertheless, we use a Euclidean distance value equal to $Dist(q.\lambda, L_{last})$ as the half window size in the equivalent window query. This justified by the fact that any place out of the window thus determined must result in a larger matching distance than does L_{last} . In this sense, our algorithm does not need to visit any farther places out of the window. On the other hand, we cannot reduce the window size to a value less than $Dist(q.\lambda, L_{last})$ because this distance itself can be a Euclidean distance if all involved places and $q.\lambda$ are in a same straight line.

Aboulnaga and Aref [2] proposed a cost model for window query processing in linear quadtrees. Given a query window W and a quadtree T , the model estimates the query cost by recursively counting the quads that overlap or are enclosed by W . This model can be employed here to estimate the IO cost our algorithm incurs in searching for trajectories with minimum match distance.

Next, we elaborate on how to estimate the minimum match distance $Dist(q.\lambda, L_{last})$ since it determines the window query size.

Let K be the total number of keywords in the entire space of interest, C be the maximum number of places per trajectory, and Q be the number of keywords in q , i.e., $Q = |q.\Psi|$. Our analysis needs the information about the trajectory places distribution in the space, as well as the keywords distribution on all trajectory places. Both distributions can be very complicated due to many hard-to-describe factors including environment and humans. We hereby make two simplifying assumptions. We assume there are w keywords on average per trajectory place, and no keyword is repeated across places within a same trajectory.

We count how many places our algorithm visits on the returned trajectory, i.e., the one with the minimum match distance $\text{Dist}(q.\lambda, L_{last})$. The counting starts at the first place where at least one required keyword in $q.\Psi$ is included, denoted as L_s , and ends at place L_{last} . Note that both L_s and L_{last} must have at least one required keyword in $q.\Psi$. We use $\hat{P}r(i)$ to denote the probability that L_{last} is the i -th place inclusively from L_s .

The probability of a single place containing the query words $q.\Psi$ is computed by

$$\begin{aligned}\hat{P}r(1) &= pr(q \in L.\Psi) = \prod_{w \in q.\Psi} (pr(w \in L.\Psi)) \\ &= \prod_{w \in q.\Psi} (1 - pr(w \notin L.\Psi_i)^{|L.\Psi|}) \\ &= \prod_{w \in q.\Psi} (1 - (1 - pr(w))^{|L.\Psi|}), \quad (1)\end{aligned}$$

where q is the query, $L.\Psi_i$ ($i \in [1, |L.\Psi|]$) is a word in $L.\Psi$, and $pr(w)$ is the probability that a word in a place L is the query word w .

We say that i places “jointly” contain the query words if 1) the i places cover the query words, 2) the first place and the last place must contain some query keywords, and 3) none of proper subsets of the i places contain all the query words. We denote the probability that i places “jointly” contain the query words by $\hat{P}r(i)$. To compute it, we first compute the probability that a subset of the i places contain the query words $pr(i)$, which can be computed as we do in Equation 1, that is,

$$pr(i) = pr(q \in \cup_{j=1}^i L_j) = \prod_{w \in q.\Psi} (1 - (1 - pr(w))^{i \cdot |L.\Psi|})$$

We next compute the probability that each place in a subset of the i places contains all the query words.

$$p_1(i) = \sum_{j=1}^i \binom{i}{j} \hat{P}r(1)^j * (1 - \hat{P}r(1))^{i-j}$$

where $\hat{P}r(1)^j$ is the probability that each of the individual j places contains all the query words, and $(1 - \hat{P}r(1))^{i-j}$ is the probability that each of the other $i - j$ places does not contain all the query words.

We next compute the probability that a proper subset of the i ($i > 2$) places jointly contains the query words such that the subset does not contain the first and the last places of the i places and none of single places contains all the query words.

$$p_2(i) = \sum_{j=2}^{i-1} \left(\binom{i}{j} - \binom{i-2}{j-2} \right) \hat{P}r(j) * (1 - Pr(i-j))$$

where $\hat{P}r(j)$ is the probability that j places of the i places jointly contain the query words, and $(1 - Pr(i-j))$ is the probability that the other $i - j$ places do not contain the query words.

Finally, we are ready to compute $\hat{P}r(i)$.

$$\hat{P}r(i) = pr(i) - p_1(i) - p_2(i) \quad (2)$$

As an example, the probability that two places L_1 and L_2 jointly contain the query words $q.\Psi$ is $\hat{P}r(2) = pr(2) - p_1(2) = pr(2) - 2\hat{P}r(1) * (1 - \hat{P}r(1)) - \hat{P}r(1)^2$.

Consequently, the expected number of places to visit is $\sum_{i=1}^{C-1} i \cdot \hat{P}r(i)$. Assuming that the average segment length of all trajectories is len , the expected distance from place L_s to place L_{last} is $len \cdot \sum_{i=1}^{C-1} i \cdot \hat{P}r(i)$.

Finally, we estimate the Euclidean distance between query location $q.\lambda$ and place L_s , i.e., $\text{Dist}(q.\lambda, L_s)$. Suppose there are Y trajectories in the entire space, which results in $Y \cdot C$ places in total. The average Euclidean distance between two adjacent places is $L/\sqrt{Y \cdot C}$, where L is the side size of the entire space. On average we need to visit $\lceil K/w \cdot Q \rceil$ places to see a required keyword in $q.\Psi$. As a result, $\text{Dist}(q.\lambda, L_s)$ is approximated by $L/\sqrt{Y \cdot C} \cdot \lceil K/w \cdot Q \rceil$.

To put it altogether, $\text{Dist}(q.\lambda, L_{last}) \approx L/\sqrt{Y \cdot C} \cdot \lceil K/w \cdot Q \rceil + len \cdot \sum_{i=1}^L i \cdot \hat{P}r(i)$. As mentioned above, this distance and the query location $q.\lambda$ together determine the window query whose cost can be estimated using the model proposed by Aboulmaga and Aref [2].

4.3 Computing Match Distance of a Trajectory

We present algorithm Match for searching the minimum match of a selected trajectory to a query, and computing the match distance. Match is invoked by algorithm IE and our baseline algorithms.

Given a trajectory $\mathcal{T}\mathcal{R} = L_1, \dots, L_n$ and a query q , a naive approach to finding the minimum match is to check all possible sub-trajectories in $\mathcal{T}\mathcal{R}$. For each sub-trajectory, we check if it is a match of the query q ; if it is, we compute the match distance. Finally, we get the minimum match distance. The time complexity of the naive approach is $O(|\mathcal{T}\mathcal{R}|^2)$.

We proceed to develop an approach with $O(|\mathcal{T}\mathcal{R}|)$ complexity based on the principle of divide and conquer and the idea of dynamic programming. Specifically, we divide the problem into sub-problems, each of which is to search the minimum match starting from a place in a trajectory $\mathcal{T}\mathcal{R}$. At each place, we check whether query q can be matched by a sub-trajectory starting at the place. Here a key idea is that we reuse the computation of the sub-problem of finding the minimum match sub-trajectory starting at the preceding place for processing the sub-problem of finding matching sub-trajectory starting at the current place. After we process all the sub-problems, we will find a minimum match, if any.

We now introduce lemmas required for developing the algorithm. Based on Definition 3, we have the following proposition.

Proposition 1: If a sub-trajectory $\mathcal{T}\mathcal{R}.L_s^e$ from place L_s to place L_e matches q , then any sub-trajectory containing $\mathcal{T}\mathcal{R}.L_s^e$ matches q . If a sub-trajectory $\mathcal{T}\mathcal{R}.L_s^e$ is not a match of q , then any sub-trajectory of $\mathcal{T}\mathcal{R}.L_s^e$ is not a match. \square

Lemma 3: If a sub-trajectory $\mathcal{T}\mathcal{R}.L_s^e$ is a minimum match of a query q , and sub-trajectory $\mathcal{T}\mathcal{R}.L_{ps}^{ed}$ is a match of query q such that $ps \leq s$ and $ed \geq e$, then $\text{matchDist}(q, \mathcal{T}\mathcal{R}.L_s^e) \leq \text{matchDist}(q, \mathcal{T}\mathcal{R}.L_{ps}^{ed})$.

Proof Sketch: We can prove the lemma by the distance triangle inequality. The distance between q and L_s must be smaller than the sum of the distance between q and L_{ps} and the distance between L_{ps} and L_s . \square

Based on Lemma 3, we have the following proposition.

Proposition 2: If a sub-trajectory $\mathcal{T}\mathcal{R}_1$ is contained by sub-trajectory $\mathcal{T}\mathcal{R}_2$, the match distance of $\mathcal{T}\mathcal{R}_1$ to query q is smaller than that of $\mathcal{T}\mathcal{R}_2$. \square

Lemma 4: Let sub-trajectory $\mathcal{T}\mathcal{R}.L_s^e$ be a match of query q . The maximal distance of all places in $\mathcal{T}\mathcal{R}.L_s^e$ to query q , i.e.,

Procedure Match(query q , trajectory \mathcal{TR} , distance ξ)

```

1 mDist  $\leftarrow \infty$ ; ts  $\leftarrow \infty$ ; te  $\leftarrow \infty$ ;           // Result variables
2  $C \leftarrow$  an array of  $|q.\Psi|$  elements of 0;           // used as the
   counter for each query word
3 for each word  $w$  in  $L_1.\Psi$  do  $C[w] \leftarrow C[w] + 1$ ;
4  $ll \leftarrow 1$ ;                                     // the last scanned place
5  $b \leftarrow 1$ ;
6 while  $ll \leq n$  do
7   ( $ism$ , mDist, ts, te)  $\leftarrow$  IsMatch( $q$ ,  $C$ ,  $b$ ,  $ll$ , mDist);
8   if  $ism$  then
9     for each word  $w$  in  $L_b.\Psi$  do  $C[w] \leftarrow C[w] - 1$ ;
10     $b \leftarrow b + 1$ ; continue;
11     $ll \leftarrow ll + 1$ ;
12    if  $\text{Dist}(q, L_{ll}) > \xi$  then
13       $b \leftarrow b + 1$ ;
14       $C \leftarrow 0$ ;                                // for all elements of  $C$ 
15      continue;
16    for each word  $w$  in  $L_{ll}.\Psi$  do
17       $C[w] \leftarrow C[w] + 1$ ;
18    if  $\min(\text{Dist}(q, L_b), \text{Dist}(q, L_{ll})) + \sum_{j=b}^{ll-1} \text{Dist}(L_j, L_{j+1}) > \xi$  then
19      for each word  $w$  in  $L_b.\Psi$  do  $C[w] \leftarrow C[w] - 1$ ;
20       $b \leftarrow b + 1$ ; continue;
21    ( $ism$ , mDist, ts, te)  $\leftarrow$  IsMatch( $q$ ,  $C$ , mDist,  $b$ ,  $ll$ );
22    if  $ism$  then
23      for each word  $w$  in  $L_b.\Psi$  do  $C[w] \leftarrow C[w] - 1$ ;
24       $b \leftarrow b + 1$ ;
25    if  $ll = n$  and not( $\forall w \in q.\Psi, c[w] > 0$ ) then
26      break;                                         // no remaining matches
27 return (mDist, ts, te);

```

Procedure IsMatch(q , C , ts, te, mDist)

Input: query q , counter vector C , start place ts, end place te, match distance mDist)

Result: ism , mDist, ts, te

```

1 if  $\forall w \in q.\Psi, C[w] > 0$  then                       // it is a match
2    $md = \min(\text{Dist}(q, L_{ts}), \text{Dist}(q, L_{te})) + \sum_{j=ts}^{te-1} \text{Dist}(L_j, L_{j+1})$ ;
3   if mDist  $> md$  then mDist  $\leftarrow md$ ;
4   return ( $true$ , mDist, ts, te);
5 return ( $false$ );

```

$\max_{i \in [s, e]} \text{dist}(q, TR.L_i)$, is a lower bound of the match distance between the sub-trajectory and q .

Proof Sketch: The proof can be established based on triangle inequality. \square

The pseudocode of the algorithm is outlined in Procedure Match. The algorithm takes in three arguments, a query q , a trajectory \mathcal{TR} , and the match distance of the current k th result. It uses a variable mDist to keep track of the current minimum match distance, and ts and te to track the start place and end place, respectively, of the corresponding minimum match (line 1). It uses an array C to keep track of the number of occurrences of query keywords (in query q) in a sub-trajectory (line 2). It uses a variable b to represent the start place of a sub-trajectory, and a variable ll to represent the end place of a sub-trajectory. The algorithm initializes array C with the occurrences of query keywords in location L_1 (line 3).

For each place L_{ll} , Procedure Match searches for a match for sub-trajectories from L_b to L_{ll} (lines 7–24). Procedure Match scans places to the right of L_b to see whether a sub-trajectory starting from L_b exists to match query q (lines 7–17). During the scan, Match updates the counter for each query keyword when it encoun-

ters a new place L_{ll} (lines 9–10).

Based on the minimum match distance ξ of the current k th result, we develop two pruning strategies.

Pruning 1: If we encounter a place L_{ll} (line 12) such that the distance $\text{Dist}(L_{ll}, q)$ is larger than the minimum match distance ξ of the current k th result, any sub-trajectory containing L_{ll} cannot be a result according to Lemma 4. Hence, any sub-trajectory starting from a place between current L_b and L_{ll} cannot be a top- k result and we will skip to the next point L_{ll+1} to search sub-trajectory starting from L_{ll+1} (line 15), which is equivalent to invoke procedure Match to find the minimum matching for sub-trajectory starting from L_{ll+1} .

Pruning 2: If $\min(\text{Dist}(q, L_b), \text{Dist}(q, L_{ll})) + \sum_{j=b}^{ll-1} \text{Dist}(L_j, L_{j+1})$ is larger than the match distance ξ , sub-trajectories starting from b to the right cannot be a top- k result (due to triangle inequality). Hence, we move to the next start place L_{b+1} (line 20).

The algorithm invokes procedure IsMatch (to be explained shortly) to check whether a sub-trajectory starting from L_b and ending at L_{ll} is a match of query q and to compute the match distance for a match (lines 7 and 21).

Pruning 3: If we find a match, we stop scanning further to the right (lines 10 and 24). This is because the sub-trajectories generated by further scanning contain the match sub-trajectory from L_b to L_{ll} , and thus will have larger match distance than that of the current one according to Proposition 2.

If we find a match, we eliminate the contribution of place L_b from C by reducing the counter $C[w]$ by 1 if word w appears in $L_b.\Psi$ (lines 9 and 23). After the elimination, C only records the frequencies of query keywords in sub-trajectory from L_{b+1} to L_{ll} . This enables us to reuse the computation at L_b to search matching sub-trajectory starting at L_{b+1} . Any sub-trajectory between L_{b+1} and L_{ll-1} must not be a match since they are contained by the sub-trajectory from L_b to L_{ll-1} , which is not a match. Hence, to find match sub-trajectory starting from L_{b+1} , we do not need to check these sub-trajectories. Instead, we check the sub-trajectories starting from L_{b+1} and ending at L_{ll} (line 7), and beyond if required (lines 11–24). In other words, we only need to scan from location ll , rather than the start location L_{b+1} , due to reusing the computation at L_b .

Pruning 4: If the sub-trajectory from L_b to the last place L_n cannot match query q , the algorithm terminates (lines 25–26) since any sub-trajectory of the sub-trajectory from L_b to L_n cannot match q according to Proposition 1.

We proceed to present Procedure IsMatch. If every query keyword in q is included in the sub-trajectory from ts to te (line 1), the sub-trajectory matches query q , and the Procedure computes the match distance md (line 2), and updates mDist with md (line 3).

The correctness of the algorithm is obvious: If there exists a minimum match in TR for query q , the match must start with a place in TR , our algorithm is able to find the minimum match starting from each place, and thus is able to find a minimum match if there is one.

Complexity: Procedure Match is a linear time algorithm, and its complexity is $O(|\mathcal{TR}|)$. Note that the words of each location are processed twice at most (once as the end of a sub-trajectory and the other as the head). Two tricks in procedure Match are essential to achieve the linear complexity: 1) we divide the task to sub-problems of finding the minimum match starting from each place; and 2) we are able to reuse the computation for the sub-problem in the preceding place.

5. EXPERIMENTS

We conduct extensive experiments on real trajectory datasets to study the performance of the proposed index B^{ck} -tree for answering TkSK queries. We build our proposed index in BerkeleyDB. In the following experiments, our approach is compared with four baseline algorithms, including IF, RT, IRT and ITB-tree. ITB-tree is presented in Appendix B.

5.1 Experimental Settings

We crawl three real spatial trajectory datasets, located in US, France and Germany, respectively, from online travel route sharing web sites^{1,2}. In the US dataset, there are 12,832 trajectories and each trajectory contains around 60 locations. France dataset contains 27,689 trajectories and each trajectory contains around 78 locations. The Germany dataset contains 40,000 trajectories and each trajectory contains an average of 40 locations. We use a real question and answer dataset to attach text to the locations in each trajectory. The dataset is publicly available from Yahoo! Webscope and contains 3,895,298 questions and their answers (Q&As), written in English. Dataset France and Germany are generated by randomly selecting a question for a location in the France and Germany trajectories. For the US dataset, we attach both a question and its answer to the locations in the US data. That is, the trajectories in the US dataset are associated with much more keywords than those in the other two datasets.

In addition to the data from online route sharing web sites, we also generate a real trajectory dataset from Flickr. We retrieve photos in New York City with shooting time, geo-location and descriptive tags from the same user and used them to generate trajectories based on the approach [19]. This dataset contains 19,104 trajectories and each trajectory contains around 4 locations.

The detailed statistics of the three generated datasets are given in Table 1.

	US	France	Germany	Flickr
#traj	12,832	27,689	40,000	19,104
#location	760,516	1,608,412	1,314,243	55,059
#word	26,792,407	9,098,284	5,620,720	2,654,477
#distinct-word	452,734	244,779	164,882	58,917

Table 1: Datasets statistics

In order to evaluate the scalability, we also generate datasets with different number of trajectories and different number of locations per trajectory by sampling the Germany dataset. The number of trajectories increases from 10K to 40K and the number of locations in each trajectory increases from 50 to 200 respectively. We list the settings in Table 2, where the default values are shown in bold.

Parameter	Setting
Datasets	US, FR, GM, Flickr
# of queries	50
k in TkSK query	5 , 10, 15, 20, 25
# of keywords in TkSK query	2, 3 , 4, 5
# of segments per quad cell	400, 600, 800 , 1000, 1200

Table 2: Experimental parameters and settings

As shown in Table 2, for each of the dataset we randomly generate a set of 50 queries and we report the average running time. I/O cost is not reported in the experiments because inverted file,

¹www.bikely.com

²www.gpsies.com

R-tree and BerkeleyDB have different file I/O mechanisms and it is difficult to find an appropriate and fair comparison method in terms of I/O cost. In the experiments, we vary the number k in the TkSK query from 5 to 25. To study the effect of the number of query keywords, we vary it from 2 to 5. Recall that our indexing approach relies on a grid partitioning of the spatial spaces. We also investigate the performance implications of different partitioning granularities. In particular, we vary the number limit of trajectory segments per cell from 400 to 1200. All the algorithms including the baselines are implemented in Java and run on a server installed with Centos operating system.

5.2 Query Performance

5.2.1 Effect of k in TkSK queries

In the first set of experiments, we fix the number of query keywords at 3 and study the effect of k in the top- k queries. We plot the average running time on the four real datasets in Figure 4. We notice that ITB incurs much higher cost than the other indexes. For instance, in the US dataset, the running time of ITB is about 3-6 times higher than IF and more than 10 times higher than our approach using B^{ck} -tree. ITB’s relatively low performance is attributed to two reasons. First, ITB indexes locations rather than trajectories. Second, a leaf node in ITB only contains the locations from the same trajectory. Hence, the ITB index contains much more nodes than do the other indexes. In order to make the figures more presentable, we do not present the results of ITB in the figures in this section.

Figure 4 shows that our indexing approach significantly outperforms the other three baseline approaches in all datasets. Note that y-axes are in logarithmic scale. B^{ck} -tree is usually around 1-2 times faster than IRT, the best baseline among the four baselines. Since IF finds all the trajectories that match the query, the running time remains constant for all values of k . The other three methods, on the other hand, incur higher cost as k increases. This is expected since they use the match distance of the k th trajectory as the pruning condition. We observe that IRT performs better than RT on datasets US, GM and FR while IRT and RT perform almost the same on dataset Flickr. IRT uses the IR-tree [10] to prune search space utilizing both spacial information and text information. IRT is effective on US, GM and FR, in which trajectories are distributed over a whole country, and thus the overlap among the MBRs of trajectories is relatively small although IRT takes a whole trajectory as an object. On the three datasets, RT is worse than IRT since RT is based on the R-tree and only uses spatial information to prune search space. However, trajectory data from Flickr is from a city, and simply treating a whole trajectory as an object yields very high overlap between MBRs and thus degrades the pruning power of text information of the IR-tree used in the IRT algorithm. The overlap between MBRs also explains why RT performs poor on dataset Flickr.

5.2.2 Effect of the number of query keywords

Next, we study the query performance when varying the number of query keywords from 2 to 5. The results are presented in Figure 5. The y-axes are also in logarithmic scale. Again, our approach provides results with the best running time over all the three datasets, and it runs 1-2 times faster than the best baseline, IRT. For IF, we observe that the more keywords are queried, the faster the results are returned. This is because IF has more query keywords to do the filtering, and IF compute the match distance for fewer trajectories that cover the query keywords. For the other tree-based approaches, more query keywords require more I/O cost to read the posting lists, and thus the running time increases slightly.

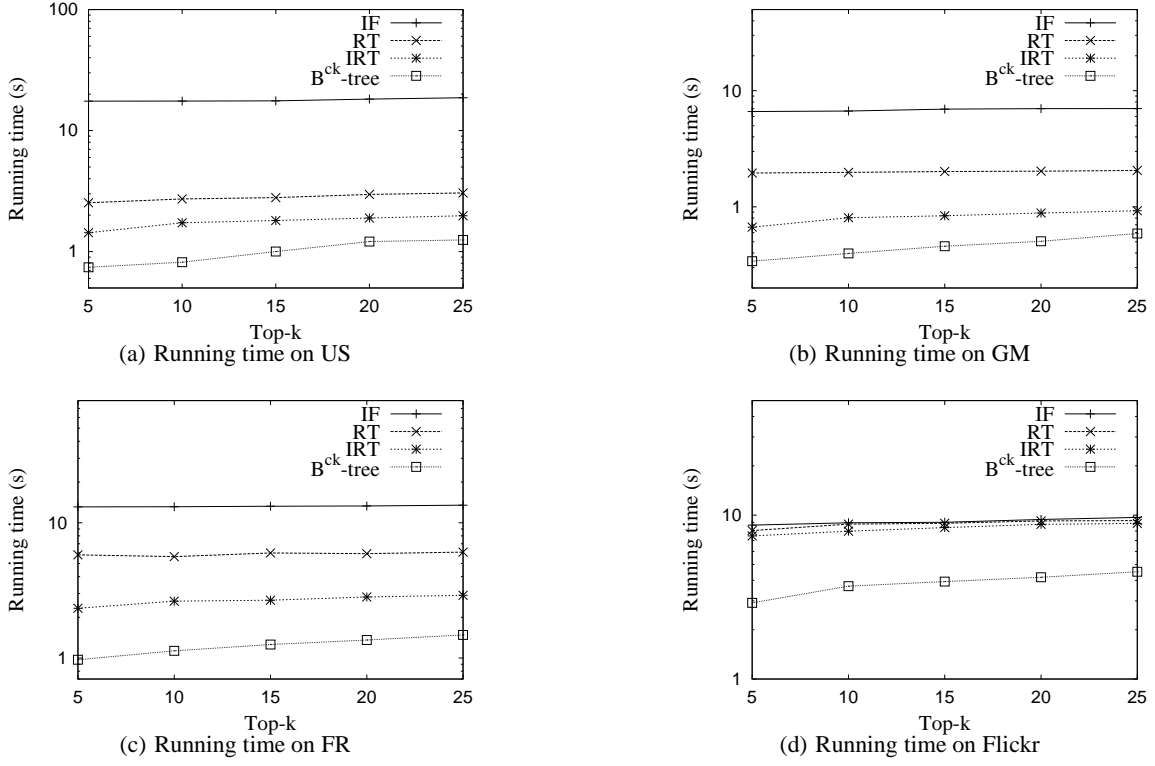


Figure 4: Varying k in TkSK queries

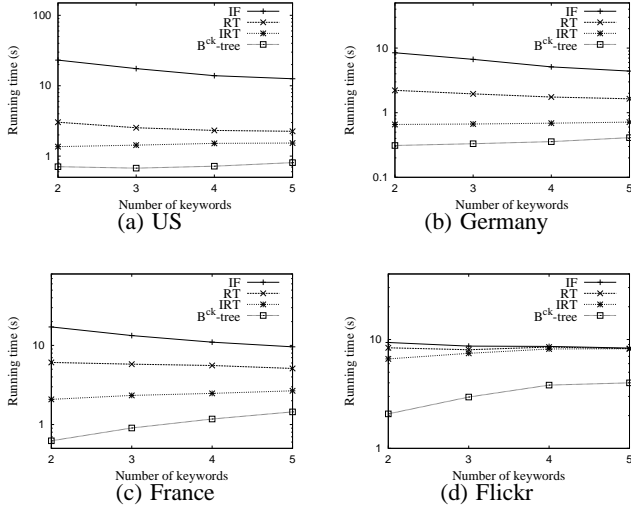


Figure 5: Varying the number of query keywords

5.2.3 Effect of partition granularity

We now proceed to study the query performance of the proposed index with regard to the partition granularity. Recall that we set a limit for the number of trajectory segments in each cell. A cell splits into 4 sub-cells when the number of segments exceeds the limit. In this experiment, we vary the number limit from 400 to 1200. The results of running time and I/O cost are shown in Figure 6. From the figure, we can conclude that our approach is not sensitive to the partition granularity. With finer partition, the performance

slightly degrades because more cells are scanned but few additional trajectories are pruned. However, this performance degradation is so small that it is negligible. In particular, when varying the limit from 400 to 1200, the running time degradation is only 0.03s.

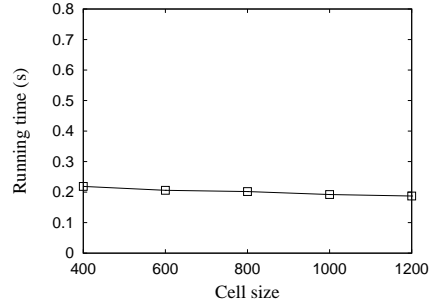


Figure 6: Varying the number limit of trajectory segments in a cell

5.2.4 Scalability

Finally, we evaluate the scalability. In this experiment, we report two sets of results. In the first set, we fix the number of locations in each trajectory at 50 and vary the number of trajectories from 10K to 40K. In the second one, we use one datasets with 20K trajectories and vary the number of locations in each trajectory from 50 to 200. The running times are shown in Figure 7. As expected, all of the four methods take linear/sublinear time. We also notice that the proposed method B^{ck} -tree scales much better than do the other methods when increasing the number of trajectories.

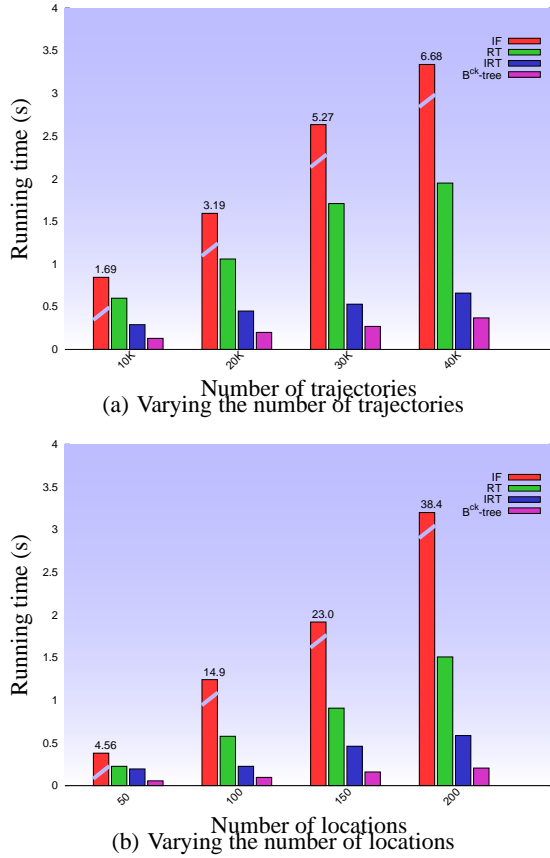


Figure 7: Scalability results

6. RELATED WORK

Trajectory Query

To the best of our knowledge, no work has considered answering the proposed TkSK queries for trajectory data.

Related to the TkSK queries is the keyword range queries supported in some online GPS trajectory sharing applications, e.g., Mountain Bike (www.bikly.com), GPS sharing, etc., in which users can share, browse and search GPS trajectories. They allow users to specify a region and a set of keywords, and return the trajectories that are inside the query region and contain the set of query keywords. However, the algorithms used are not publicized.

Existing work on spatial-temporal trajectory indexing schemes [6, 11, 22, 26, 27] clearly focuses on trajectories without text data. These index structures are usually designed for keep tracking of moving objects. A number of algorithms have been proposed to process different types of spatial-temporal queries, such as k nearest neighbor queries (e.g., finding the k -closest objects with respect to a given point at a given time), range queries (e.g., finding all objects within a given area), and complex spatial pattern queries [9, 15, 28].

A number of similarity functions and algorithms have been developed to compute the similarity between trajectories/time series data, e.g., [3, 7, 29]. Also, there exist work on trajectory pattern discovery [20], clustering trajectories [18], and finding significant locations from trajectories [4].

Spatial Keyword Search

Zhou et al. [32] handle the problem of retrieving web documents relevant to a keyword query within a pre-specified spatial region. Similar problem is also considered by Chen et al. [8] and Hariharan

et al. [17]. These proposals use loose combinations of an inverted file and a spatial index (e.g., R-tree). The query processing in these proposals occurs in two stages: One type of indexing (e.g. inverted list) is used to filter web document in the first stage, and then the other index (e.g. R-tree) is employed, or the vice versa. This index has the disadvantage that it cannot simultaneously prune the search space using both keywords and spatial distance.

Felipe et al. [12] propose a novel index structure called IR^2 -tree that augments an R-tree with signatures. For the first time, the new hybrid index structure enables to utilize both spatial information and text information prune search space at query time, which advances the state-of-the art in spatial-keyword query processing. However, this proposal suffers from the crucial limits of signature files (e.g., the number of false matches is linear in the collection size [33]). Further, the IR^2 -tree faces the challenge of whether the signatures possess enough pruning power to offset the extra cost incurred by the taller trees that result from inclusion of signatures.

The hybrid index structure that combines R^* -tree and bitmap indexing is developed to process a new query called m -closest keyword query [30] that returns the closest objects containing at least m keywords. This index structure exhibits the same problems as do signature-file based indexing [12].

The hybrid index structure IR-tree [10] that integrates the R-tree and inverted file enables the efficient processing of the location-aware top- k ranking query by utilizing both location and text information to prune the search space. In the IR-tree [10] the fanout of the tree is independent of the number of words of objects in the dataset, and, during query processing, only (a few) posting lists relevant to the query keywords need to be fetched. A recent proposed index named Spatial Inverted Index [24] maps each keyword to a distinct aggregated R-tree [21] that stores the objects containing the given keyword. The collective spatial keyword query [5] aims to retrieve a group of nearby objects that cover the query keywords.

None of these proposals considers trajectory data associated with text as does this paper. Moreover, these proposed hybrid index solutions are not supported by the mainstream DBMSs. In contrast, the proposed solution in this paper is ready to be implemented on the DBMSs.

Finally, note that the proposed TkSK query is complementary to the route planning queries (e.g., [25]), which return a route of places from a spatial database such that the route covers a set of query keywords and the travel distance is minimized.

7. CONCLUSION

This paper proposes a new algorithm IE for efficiently answering TkSK queries on trajectory data associated with text descriptions. The algorithm is developed based on a new hybrid index called cell-keyword conscious B^+ -tree, denoted by B^{ck} -tree. B^{ck} -tree allows us to develop algorithms that exploit both text relevance and location proximity to facilitate efficient and effective query processing. Additionally, the algorithm Match is proposed for efficiently computing the match distance between a query and a trajectory. The experimental results demonstrate that the proposed algorithm outperforms several baseline algorithms significantly and offers good scalability.

8. REFERENCES

- [1] D. J. Abel and J. L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Int. Journal of Comp. Vision, Graphics, and Image Processing*, 24:1–13, 1983.
- [2] A. Aboulmaga and W. G. Aref. Window query processing in linear quadrees. *Distributed and Parallel Databases*, 10(2):111–126, 2001.
- [3] R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In *FODO*, pages 69–84, 1993.

- [4] X. Cao, G. Cong, and C. S. Jensen. Mining significant semantic locations from gps data. *PVLDB*, 3(1), 2010.
- [5] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *SIGMOD Conference*, pages 373–384, 2011.
- [6] V. P. Chakka, A. Everspaugh, and J. M. Patel. Indexing large trajectory data sets with seti. In *CIDR*, 2003.
- [7] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD*, pages 491–502, 2005.
- [8] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD*, pages 277–288, 2006.
- [9] Z. Chen, H. T. Shen, X. Zhou, Y. Zheng, and X. Xie. Searching trajectories by locations: an efficiency study. In *SIGMOD*, pages 255–266, 2010.
- [10] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.
- [11] P. Cudré-Mauroux, E. Wu, and S. Madden. Trajstore: An adaptive storage system for very large trajectory data sets. In *ICDE*, pages 109–120, 2010.
- [12] I. De Felipe, V. Hristidis, and N. Rische. Keyword search on spatial databases. In *ICDE*, pages 656–665, 2008.
- [13] E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis. Nearest neighbor search on moving object trajectories. In *SSTD*, pages 328–345, 2005.
- [14] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, 1984.
- [15] M. Hadjieleftheriou, G. Kollios, P. Bakalov, and V. J. Tsotras. Complex spatio-temporal pattern queries. In *VLDB*, pages 877–888, 2005.
- [16] Q. Hao, R. Cai, C. Wang, R. Xiao, J.-M. Yang, Y. Pang, and L. Zhang. Equip tourists with knowledge mined from travelogues. In *WWW*, pages 401–410, 2010.
- [17] R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing spatial-keyword (SK) queries in geographic information retrieval (GIR) systems. In *SSDBM*, 2007.
- [18] J.-G. Lee, J. Han, and K.-Y. Whang. Trajectory clustering: a partition-and-group framework. In *SIGMOD*, pages 593–604, 2007.
- [19] X. Lu, C. Wang, J.-M. Yang, Y. Pang, and L. Zhang. Photo2trip: generating travel routes from geo-tagged photos for trip planning. In *ACM MM*, pages 143–152, 2010.
- [20] N. Mamoulis, H. Cao, G. Kollios, M. Hadjieleftheriou, Y. Tao, and D. W. Cheung. Mining, indexing, and querying historical spatiotemporal data. In *KDD*, pages 236–245, 2004.
- [21] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient olap operations in spatial data warehouses. In *Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases, SSTD*, pages 443–459, London, UK, UK, 2001. Springer-Verlag.
- [22] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *VLDB*, pages 395–406, 2000.
- [23] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer. Integrating the ub-tree into a database system kernel. In *VLDB*.
- [24] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvg. Efficient processing of top-k spatial keyword queries. In *SSTD*, pages 205–222, 2011.
- [25] M. Sharifzadeh, M. Kolahdouzan, and C. Shahabi. The optimal sequenced route query. *The VLDB Journal*, 17:765–787, July 2008.
- [26] Y. Tao and D. Papadias. Mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In *Proceedings of VLDB*, pages 431–440, 2001.
- [27] Y. Theodoridis, T. K. Sellis, A. Papadopoulos, and Y. Manolopoulos. Specifications for efficient indexing in spatiotemporal databases. In *SSDBM*, pages 123–132, 1998.
- [28] M. R. Vieira, P. Bakalov, and V. J. Tsotras. Querying trajectories using flexible patterns. In *EDBT*, pages 406–417, 2010.
- [29] M. Vlachos, D. Gunopoulos, and G. Kollios. Discovering similar multidimensional trajectories. In *ICDE*, page 673, 2002.
- [30] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *ICDE*, pages 688–699, 2009.
- [31] Y. Zheng, X. Xie, and W.-Y. Ma. Geolife: A collaborative social

networking service among user, location and trajectory. *IEEE Data Eng. Bull.*, 33(2):32–39, 2010.

- [32] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma. Hybrid index structures for location-based web search. In *CIKM*, pages 155–162, 2005.
- [33] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.

APPENDIX

A. DISCUSSION ON UPDATES

Deleting and replacing trajectories would seldom happen to a trajectory repository. Thus we only consider inserting new trajectories. To insert a new trajectory, the insertion algorithm first locates those cells into which the trajectory falls. After that, it checks whether the number of trajectory segments in each of the located cells is still within the limit. If it is the case, the algorithm associates the words of the trajectory with the corresponding cells according to the method discussed in Section 4.1 and inserts the entry $\langle \text{wordID}, \text{cellID}, \text{tID} \rangle$ into the B^+ -tree. For a cell where the number of trajectory segments exceeds the limit after insertion, the cell will be split into 4 sub-cells. A re-computation of the word-cell association is needed. After that, the algorithm inserts into the B^+ -tree the entries with respect to the newly created four (sub-)cells and removes the obsolete entries associated to the old cell.

B. BASELINE 4: ITB-TREE INDEX BASED ALGORITHM

The baselines TR and IRT treat each trajectory as an object to build index. The ITB-tree index treats each location of a trajectory, rather than the whole trajectory, as an object.

We proceed to briefly present an index structure, the ITB-tree (Inverted file augmented TB-tree), and the idea of an algorithm based on the ITB-tree for the TSK query. The ITB-tree is essentially a TB-tree [22] augmented with inverted files. The TB-tree [22] is proposed for indexing trajectory data without text information to efficiently support location based queries. The ITB-tree inherits the property of TB-tree [22] that is capable of preserving consecutive locations of the same trajectory in an index.

Each leaf node in the ITB-tree contains entries of the form $e = (\lambda, \psi)$, where e represents a place of a trajectory in dataset \mathcal{D} , λ is the minimum bounding rectangle (MBR), which is a point for a place, and $e.\psi$ refers to the id of the text description of the place. Each leaf node contains a pointer to an inverted file with the text descriptions of the objects stored in the node. In addition, each leaf node maintains two pointers (forward and backward) that link the leaf node to other leaf nodes that contain adjacent sub-trajectories of the sub-trajectory contained in the leaf node.

Each non-leaf node CN in the ITB-tree contains a number of entries of the form (e, λ, ψ) where e is the address of a child node of R , λ is the MBR of all rectangles in entries of the child node, and ψ is the identifier of a pseudo text description that is the union of all text descriptions in the entries of the child node. The pseudo text description is a union of the text descriptions of the children nodes. Each non-leaf node also contains a pointer to an inverted file with the text descriptions of the entries stored in the node.

We treat query q as a set of partial queries, where each partial query has a keyword in $q.\psi$ and the spatial component $q.\lambda$. For each partial query we find its nearest places incrementally using the ITB-tree index. When a trajectory is covered by all the partial queries, i.e., some place in the trajectory is retrieved as a nearby place for each partial query, we choose the trajectory as a candidate and compute the match distance of the trajectory to the query. Intuitively, the trajectory would be a good candidate of the top- k results since it contains all the query keywords and its places covering the keywords are close to the spatial component of the query. The detailed pseudo code of the partial query evaluation Algorithm can be found in our technical report.